XCPL: An Experimental Concurrent Programming Language

W. C. Athas

Department of Computer Science
California Institute of Technology

5196:TR:85

XCPL : An Experimental Concurrent Programming Language

by

w.c. athas

Oldthinkers unbellyfeel Ingsoc.

Computer Science Department
California Institute of Technology
Pasadena, Calif. 91125

Technical Report 5196:TR:85

December 7, 1985

The research described in this report was sponsored in part by a grant from Intel Corporation and in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597

You are never too old to have a happy childhood.

- Tom Robbins

1 Introduction

Experimental Concurrent Programming Language (XCPL) is a programming language based on history-dependent objects that interact solely by message passing. XCPL is both an experiment to develop a powerful programming notation for describing concurrent computations and an experiment to determine whether programs so described can be efficiently used in highly concurrent message passing systems such as ensemble machines [Seitz83]. The approach chosen is to describe a computation in terms of concurrent objects that represent the various components of some larger task. The objects work cooperatively on the common task and coordinate their efforts by message passing. A prime concern of the experiment is the compilation of XCPL programs. The goal of compilation is to keep intact the concurrent behavior expressed in the notation so that the concurrent behavior is also present in the distributed environment where the program is to be executed.

A running XCPL program contains three general types of activities: performing operations on primitive data types, creation of new objects, and message passing. Operations performed on primitive data types correspond to the usual répertoire of predefined operations such as addition, subtraction, and comparison, performed on the usual set of predefined data types, *e.g.* integers, reals, Booleans, symbols, *etc.* The creation of new objects is fundamental to amassing a collection of concurrent objects. The sending of a message corresponds to requesting a *single* service of a destination object. After the destination object has processed the message, it will send back a reply. The interim between requesting a service and receiving a reply does not necessarily "stop" the sender; rather the only atomic action associated with the send is the establishment of a reservation for the reply value. A sender may have multiple reservations outstanding and still be making progress. A sender will only wait for a reply value when progress cannot be made without the reply value, or when explicitly expressed by the programmer. This paradigm for message passing has two interpretations. Viewed purely in terms of primitive message passing operations, the paradigm cleanly captures the weak synchronization property of the message functions used in the Cosmic Cube/Cosmic Kernel experiment [CPGCC]. If the message send is viewed as a function call and the reply a return from a call, then the paradigm closely resembles the "futures" construct [Baker] as used in the language Multilisp [Halstead].

XCPL assumes that programming with concurrent objects is no more difficult, if not simpler than programming with sequential objects. The difficulties associated with formulating concurrent programs at the object level are perceived as problems largely of definition, once the rules of object interaction are understood. To support this conjecture as well as to provide a practical introduction to XCPL, four exemplary programming problems are solved in Section 3 of this paper. The examples are a (7,4) Hamming decoder, two solutions to the Eight Queens problem, Gaussian elimination, and the generation of primes by sieving. The Hamming decoder demonstrates the rudimentary composition style for concurrent object programming. The Eight Queens problem is to place eight queen chess pieces on a chessboard so that none are in jeopardy of capture. In contrast to the Hamming decoder, the Eight Queens problem involves creating new objects as well as some rather involved object interaction. The Gaussian elimination example was included primarily to balance the symbolic predilection that occasionally creeps into XCPL with a standard problem from linear

algebra. The problem is centered upon a matrix formulation, as is the Hamming decoder, but uses the matrix in a less systolic fashion than the Hamming decoder. The last example is to generate an infinite sequence of primes. This example, though simpler in formulation than the others, presents an infinite computation with an unusual object structure. The examples have been "exercised" using the XCPL engine described in Section 4 of this paper and have been demonstrated to yield the correct solutions for the given input.

For any given application to be programmed, the first step is to find a concurrent formulation using the computational model of XCPL. This step is largely application dependent and ultimately depends upon the ingenuity of the programmer. The expressive power of the XCPL computational model does however influence the formulation. The computational model of XCPL is quite permissive and engenders only abstract properties of ensemble machines. The XCPL model however, is not as permissive as the actor model [Clinger].

The framework of XCPL is intentionally kept small to keep the synthesis of XCPL programs from a programmer's formulation largely systematic. In XCPL, all objects are presumed to execute concurrently and will do so unless explicitly programmed otherwise. The XCPL compiler therefore can never discover concurrency within a program for no concurrency is ever hidden. All sequential behavior imposed on an object must be explicitly expressed by a programmer. The compiler focuses on identifying which definitions produce sequential objects and subsuming them as components of larger concurrent objects. A static analysis of the message flow between the concurrent objects is then performed to infer the logical *structure* between the communicating objects. Whereas it is not mandatory that the logical structure be accurately predicted *a priori* for a program to compile, the better the prediction, the better the runtime performance. Runtime performance therefore depends both upon the capability of the compiler to infer the logical structure from the program text, and upon the programmer to impart the logical structure of the concurrent formulation as an XCPL program. This perspective of the compilation process is the foundation for the programming examples that follow in Section 3.

How should meaning be defined?

- Niklaus Wirth

2 Syntax and Semantics of XCPL

The expressive power of XCPL is derived from a careful confluence of the control flow idioms from imperative programming languages such as ALGOL, and the message passing metaphor as developed for the Smalltalk systems [Ingalls]. The syntactic flavor of XCPL bears strong resemblance to the language Euler, an ALGOL derivative created by Niklaus Wirth and Helmut Weber [Wirth]. Like Euler, every syntactic clause, *i.e.* text separated by either a comma, semicolon, or period, must yield a value. Every value in XCPL represents an *object* and every object can be given a name.

Objects are categorized as either *primitive* or programmer defined. Primitive objects consist of the usual predefined data types such as integers, reals, symbols, and Booleans. The behavior of these objects cannot be modified by a program, hence they are history-insensitive. This property allows primitive objects to be copied without restriction, for all copies will have the same behavior. In contrast, programmer defined objects have an associated *environment* that consists of a collection of variable names along with the values bound to each of the names. The behavior of objects that have environments can be modified by changing the bindings of values to names. Since environments can be side-effected, access to environment objects is serialized. This restriction is necessary because how an object will react to a message depends upon the binding of values to names. If multiple concurrent assignments of values to a single name were allowed from different external objects, then environment objects would not be able to control access to their internal variables.

The BNF description shown in Figure 1 defines the production rules for writing well-formed XCPL programs. Starting with the topmost production, every XCPL program must start with a block clause at the outermost level. The block clause is denoted by the keyword pair `BEGIN...END` and defines a context for executing a collection of XCPL clauses, therefore every block clause will create an environment object. Blocks may be nested. Two clauses within a block environment are composed in a top to bottom, left to right fashion by placing a combinative form, called a "combinator", between the two clauses.

Every clause in an XCPL program must yield a value, however as mentioned in Section 1, a message send will generate a reservation which is a "stand-in" or surrogate value. When the reply value of the send is available, the reservation is replaced by the actual value. This property of the language requires that there be more ways to sequence a computation than are usually found in languages such as ALGOL. For example, ALGOL uses the semicolon symbol as a combinative form to specify sequential behavior. A semicolon that separates two clauses indicates that the value of the left clause is to be computed, followed by the computation of the value for the right clause. For a sequential language, additional combinators are unnecessary since strictly top to bottom, left to right execution of clauses performs as well as any other sequence, and is convenient for it follows the convention for reading English prose. This discipline for sequencing is often extended to Boolean conjunctives and the arguments of a procedure parameter list which are separated by a different combinator, *e.g.* the comma symbol.

In XCPL, the semicolon combinator of ALGOL is represented by a period symbol. XCPL includes two more combinators for expressing synchronizations that are "weaker" than the demanding of the value for the left clause before computing the value of the right clause.

⟨*program*⟩ ⟹ ⟨*block*⟩
⟨*block*⟩ ⟹ `BEGIN` ⟨*clause*⟩ {⟨*comb*⟩ ⟨*clause*⟩ }* `END`
⟨*clause*⟩ ⟹ ⟨*block*⟩ | ⟨*assign*⟩ | ⟨*keyword*⟩ | ⟨*method*⟩ | ⟨*let*⟩ | ⟨*send*⟩
⟨*assign*⟩ ⟹ ⟨*var*⟩ `:=` ⟨*clause*⟩
⟨*keyword*⟩ ⟹ ⟨*if*⟩ | ⟨*repeat*⟩ | ⟨*while*⟩ | ⟨*for*⟩
⟨*if*⟩ ⟹ `IF` ⟨*clause*⟩ `THEN` ⟨*clause*⟩ `ELSE` ⟨*clause*⟩
⟨*repeat*⟩ ⟹ `REPEAT` ⟨*clause*⟩ `UNTIL` ⟨*clause*⟩
⟨*while*⟩ ⟹ `WHILE` ⟨*clause*⟩ `DO` ⟨*clause*⟩
⟨*for*⟩ ⟹ `USING` ⟨*range*⟩ `FOR` ⟨*var*⟩ `DO` ⟨*clause*⟩
⟨*let*⟩ ⟹ `LET` ⟨*ident*⟩ ⟨*declop*⟩ ⟨*clause*⟩
⟨*declop*⟩ ⟹ `=` | `:=`
⟨*comb*⟩ ⟹ `;` | `,` | `.`
⟨*method*⟩ ⟹ `'` {⟨*fordecl*⟩ }* ⟨*clause*⟩ `'`
⟨*fordecl*⟩ ⟹ `FORMAL` ⟨*ident*⟩ `;`
⟨*var*⟩ ⟹ ⟨*ident*⟩
⟨*send*⟩ ⟹ ⟨*prim*⟩ { ⟨*selector*⟩ { ⟨*send*⟩ } }
⟨*prim*⟩ ⟹ ⟨*int*⟩ | ⟨*bool*⟩ | ⟨*char*⟩ | ⟨*selector*⟩ | ⟨*list*⟩ | `self` | `user` | Ω
⟨*selector*⟩ ⟹ ⟨*symbol*⟩ | ⟨*var*⟩ | `(` ⟨*clause*⟩ `)`
⟨*symbol*⟩ ⟹ `:`⟨*ident*⟩
⟨*range*⟩ ⟹ ⟨*send*⟩ ⟨*comb*⟩ ⟨*comb*⟩ ⟨*send*⟩
⟨*list*⟩ ⟹ `[]` | `[` ⟨*clause*⟩ {⟨*comb*⟩ ⟨*clause*⟩ }* `]`
⟨*bool*⟩ ⟹ `true` | `false`

Figure 1: BNF Description for XCPL

The "weakest" combinator is denoted by a comma and indicates that the computation for the values of the left and right clauses are independent and their evaluation may proceed concurrently. This interpretation is purely from the perspective of the programmer and is not absolute, for if there is a data dependence between the two clauses, the dependent clause will wait for the necessary value to become available. In other words, using the comma combinator will never generate a runtime error by attempting prematurely to use a value while it is still a reservation. The third combinator is denoted by a semicolon and indicates that the computation of the value of the left clause will commence before the computation of the value of the right clause. An example of where the semicolon combinator is useful is in the sending of multiple messages to a single common destination object. Since the message system guarantees preservation of message order on sending, using the semicolon combinator will ensure that messages are sent in a particular order without having to wait for each reply before starting the next send. For all three combinator, the value of the composition is the value of the rightmost clause. For the block construct, the value yielded is the value of the last clause, *i.e.* the clause closest to the `END`.

As was mentioned above, an environment object contains a set of variable names along with the values bound to each of the variables. The binding of values to variables is changed

by executing an assignment clause that has the usual interpretation of setting the contents of the variable name on the left hand side of the assignment operator to the value of the clause on the right hand side of the assignment operator. New variable names are introduced into a block environment by the LET clause. This clause looks similar to the assignment clause but is prefixed with the keyword LET . The interpretation is to add the variable name to the environment of the *immediately* enclosing block and then bind the value of the clause on the right hand side of the assignment operator to the new variable name. For LET clauses, the equal operator can be used instead of the assignment operator to denote that once the variable has been assigned its initial value, the value will never be changed.

XCPL provides special keywords for expressing the control structures found in imperative languages such as ALGOL. Although these constructs could be defined in terms of message passing, their general usefulness warrants inclusion as fundamental to the internal behavior of environment objects. Referring to the BNF description, the two iterative constructs of REPEAT...UNTIL and WHILE...DO as well as the decision construct of IF...THEN...ELSE, are well documented elsewhere in the literature of structured programming and will not be discussed here. The construct of *USING...FOR...DO* is special to XCPL in that it is a generalization of the more traditional FOR...DO construct of sequential programming. A problem with the latter FOR clause is that it was developed exclusively for the period (sequential) combinator. Since XCPL has three combinators, nominally three versions of the FOR clause would be necessary. Instead, the usage of the combinators is expanded so that when two identical combinators are sandwiched between two primitive objects, the meaning is to create the ordered collection of primitive objects within the range of the two delimiting objects. For example, `1::8` would be equivalent to `1:2:3:4:5:6:7:8` and `4,,7` would be equivalent to `4,5,6,7`. Currently the only primitive objects for which a total order is defined are the integers. Thus an ALGOL expression of "`FOR i := 1 TO 8 DO s := s + i`" would be expressed as "`USING 1..8 FOR i DO s := s + i`". Notice that "`USING 1,,8 FOR i DO s := s + i`" permits the additions to be performed in any order, however no concurrency is achieved.

Since each of the keyword clauses is an abstraction over control flow and not values, some thorny issues are raised for the USING...FOR...DO and WHILE...DO iterative clauses. For example, if the predicate of the WHILE loop is initially false, then the loop is never executed and hence no value should be associated with the clause. One solution would be to allow iteration only in the form of tail recursion. This approach is overly restrictive since it introduces unnecessary additional variable names that are intrinsic to the iterative constructs. Therefore, to include these nice constructs and at the same time keep the semantics simple, the iterative constructs are given the following interpretation. An iterative construct initially has the value of undefined, or Ω, and for each each iteration completed, the value of the previous iteration is discarded. If the loop is executed zero times, then no value is discarded and the value of the clause is Ω. Note that the value of Ω can be tested for, though a programmer advisory is issued to programmers who write code that depends upon this interpretation of zero iterations.

XCPL parts company with both ALGOL and Euler by subsuming the notion of a procedure or function call into XCPL's message passing metaphor. Rather than calling a function, a message is sent to an environment and the inception of the message at the destination environment causes a method environment to be created. Methods are the 'reactive' values in XCPL in that they react with other values to create new objects. Referring to the BNF description, a method is defined simply by enclosing a clause in a matched pair of single quotes (` '). In order for a method to react with external values, *i.e.* the contents of a

message, formal variables are declared inside the method definition so that values may be passed into the method environment. Such variables are declared with the FORMAL keyword.

Whenever a clause is not recognized as one of the above types, it is assumed to be a send clause. To understand how send clauses work, it is helpful to examine how such expressions are parsed. Send clauses are parsed from left to right using a format of:

object selector argument

Selector must always evaluate to a symbol. In XCPL, symbols are one of the predefined data types and are recognized as belonging to a special set of single character keystrokes, *e.g.* +,-,*, and /, or as a variable name that is prefixed by a colon character (':'). The colon is necessary to ensure that the sending object does not attempt to interpret the name following the colon as a variable within its scope. The use of colon is akin to the quote operator found in LISP. For the case where *object* is a primitive object, *selector* corresponds to an XCPL defined operation that can be performed on the primitive data type. *Argument* is necessary for those cases where the primitive operation is dyadic.

The more interesting case is where *object* is an environment object. The interpretation for such send clauses is to look up *selector* as a variable name in the environment of *object*. If the value bound to *selector* is a method definition, then a method environment is created whose initial environment is the contents of the message, that is, *argument*. The resulting object is then executed. If the value bound to *selector* is not a method definition, then the message is left pending until the variable names becomes bound to a method value. Since every clause must yield a value, the value of the send clause is the value computed by the invoked method. Thus the resultant value of the method clause must be returned to the sender.

A simple example of a send clause would be "`user :print "hello"`". The destination object `user` is one of two predefined objects in XCPL. The other predefined object is `self`, which always refers to the immediate environment object. The object `user` exists on the periphery of the XCPL system and provides an interface to "real" objects in the outside world, *e.g.* computer terminals, disk files, and laser printers. By sending the `user` object the message selector `:print` with an argument of the string `"hello"`, the intent is that the string will be displayed on some output device, for example a computer terminal. Referring back to the explanation of the `USING` clause, the difference between the combinators can now be readily demonstrated. A clause of the form "`USING 1..8 FOR i DO user :print i`" will produce an output of "`1 2 3 4 5 6 7 8`". If the comma form, "`USING 1,,8 FOR i DO user :print i`", is used, then some permutation of 1 through 8 will be output, for example, "`3 6 2 8 5 7 1 4`". Note that the semicolon form would produce the same output as the period form, with the only discernible difference being that the semicolon form would be quicker.

To simplify the use of send clauses, the XCPL parser permits the appending of additional *selector* and *argument* pairs so that a sequence of message sends may be activated from left to right. Note that the interpretation is to start a send using the first three values of the clause, wait for the reply value, and then send the next two values of the send clause to the result of the first evaluation. For example, "`self :fac n + 1`" would cause the object `self`, *viz.* the current object environment, to be sent a `:fac` message with an argument of `n`. The reply value from the `:fac` message would then be sent a `+` message with argument of `1`. This parsing mode will continue until either a combinator is encountered or an open parenthesis is encountered. Matched pairs of parentheses may be used for any of three reasons:

1. To change the evaluation order for a send clause.

2. To embed a clause within a send clause.

3. To denote a send clause that requires no argument (monadic).

As was mentioned at the beginning of this section, XCPL is a confluence of the imperative idioms of ALGOL and the message passing metaphor. In order to reflect the confluence in the syntax, the strategy is to scan for keywords, and if none are found, assume the clause is a send clause. Once in a send clause, the embedding of a control clause may be desirable, hence the inclusion of the second use of parentheses. The third use for parentheses arises from the interleaving of monadic selectors with dyadic ones. Without knowing beforehand which selectors are monadic and which are dyadic, the parser cannot be expected to find the correct interpretation. Hence selectors are assumed to be dyadic with monadic ones clearly delineated by enclosing the argumentless send clause in parentheses.

An important property of the XCPL computational model is that messages may incur arbitrary delay in transmission. To incorporate this property into the language, the only atomic action associated with a message send is the establishment of the reservation for the reply value. If a programmer were to use the comma combinator exclusively, then the compiler would introduce sequencing only when necessary to satisfy the data dependences between clauses. Additional sequencing is explicitly introduced by the programmer using the semicolon and period combinators. Such a compilation strategy always provides the programmer with maximal concurrency but allows additional "pruning" of the concurrency to be explicitly incorporated. An interesting offshoot experiment would be to make the reservations for reply values part of the value domain of XCPL. Such a modification would allow reservations to be passed as arguments of messages with actual values used only when absolutely necessary, *e.g.* performing primitive object operations.

Example is always more efficacious than precept.

- Samuel Johnson

3 Four XCPL Programming Examples

For a gentle introduction to the XCPL notation without straying too far from the ALGOL style of programming, consider the program of Figure 2. The task at hand is to compute the inner product of two integer vectors of equal length. Once the inner product has been computed, the result value is to be printed out. The chosen solution strategy is to define a method named `ip` that accepts two formal arguments, `u` and `v`, which are presumed to be lists of equal length. Once the method is activated, the list `u` is sent a `:length` message. The combinator of choice here is comma since the message send can proceed concurrently with the declaration for the variable `s`. Once the length value is received as a reply, it is bound to the variable `n`, after which the `USING` clause can be executed.

The `USING` clause will execute all n instances of its loop body concurrently since the range from 1 to n is defined using the comma combinator. Indexing into the two lists is done by sending each list an `:index` message with index value `i`. As the reply values are received, they are pairwise multiplied and summed into `s`, thereby computing the inner product function and storing the result in the variable `s`. By the semantics of the block clause, it is unnecessary to explicitly state `s` as the final clause, for the value of the `USING` clause will be the last value assigned to `s`.

In order to "call" the `ip` method, it is necessary to send the environment of the outer block an `:ip` message along with two lists as arguments. This is accomplished by a send clause whose destination object is `self` and whose argument is a list that internally consists of two lists. The message will cause the environment object to look up the symbol `:ip` and find that it is bound to a method definition. A method environment is then created to execute the method. The value replied by the method environment is `s`, the inner product of the two list arguments. Once the inner product reply is received, a second send operation is started that will send the computed inner product value to the predefined object `user` as the argument of the `:print` message for output display.

The program of Figure 2 attempted to demonstrate ALGOL style programming in XCPL. The strategy in this first example was to define an inner product method for the environment created by the outer block clause and then send it two vectors to multiply together. Another strategy which some might find more natural would be to define and instantiate a vector object which "knows" how to compute the inner product of itself with another vector. Thereby the operation of inner product is *associated* with vector objects, rather than with an arbitrary block environment. A reformulation of the program of Figure 2 is shown in Figure 3. As before there is the definition of the `ip` method and the send to the object `user`, however the `ip` method has now been moved inside another method called `vector` and the send clause now contains an additional nested send clause.

```
BEGIN
  LET ip = 'FORMAL u; FORMAL v;
    BEGIN
      LET n = (u :length),
      LET s := 0;
      USING 1,,n FOR i DO s := s + ((u :index i) * (v :index i))
    END';
  user :print (self :ip [[1,2,3],[4,5,6]])
END
```

Figure 2: XCPL Inner Product Program

```
BEGIN
LET vector = 'FORMAL u;
  BEGIN
    LET ip ='FORMAL v;
      BEGIN
        LET s := 0;
        USING 1,,n FOR i DO s := s + ((u :index i) * (v :index i))
      END',
    LET n = (u :length);
    self
  END';

  user :print ((self :vector [1,2,3]) :ip [4,5,6])
END
```

Figure 3: Inner Product as a Method of Object Vector

The `vector` method takes a single formal parameter `u` which is intended to be a list. When the `vector` method is invoked, an environment object is created by the `BEGIN...END` pair. Internal to this environment object, the `ip` method is defined concurrently with the assignment of the length of `u` to the variable `n`. The value yielded by the `vector` block clause is set to `self`. Previously `self` had been used so that a messages could be sent to the enclosing environment. Now `self` is used so that *another* object may send the current environment a message. To see how this technique is used, consider the send clause for `user`. The nested send clause: "`self :vector [1,2,3]`", will cause the environment of the outer block to be sent a `:vector` message which in turn will cause the `vector` method to be invoked. Once invoked, `[1,2,3]` is bound to `u`, the length of `u` is computed, the `ip` method is defined and the value of the instance of the environment object is replied. The replied environment object value is then sent an `:ip` message with an argument of `[4,5,6]`. The block environment created by the `vector` method receives an `:ip` message which it can react to via the `ip` method definition. The `ip` method is the same as was described for Figure 2.

Although for this second example, the use of the object `vector` is slightly contrived, it does serve as the foundation for a set of operations defined on vectors. To continue on this theme, consider the problem of providing one bit of error correction for a (7,4) Hamming code [McEliece]. Given the H matrix, the first step is to compute the syndrome vector. Assuming that the H matrix is:

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

and that the bad codeword is $y = [1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1]$, then the first step is to compute the syndrome: $s^T = Hy$. The program in Figure 3 is easily modified for this particular application and is shown in Figure 4.

In the computation of inner product, the arithmetic is now done modulo 2. Notice that the H matrix is described as a "list of lists" with each element of the outer list a `vector` object which internally holds a list. The object structure is described by the definition of the `H_matrix`, and hence, the object structure is constant. The definition of `syn` is to send to each row vector in the H matrix the bad codeword and compute the inner product of each row with the bad codeword. For both `H_matrix` and `syn`, the semantics of the comma combinator can now be deployed non-trivially to achieve three-fold concurrent activity, since in both definitions there are no dependencies between the parameters of the outer list.

By the construction of the H matrix, the value of the syndrome is the same as the column of the offending bit, and this syndrome is the value sent to `user` as the argument of the `:print` message. If the syndrome is all zeros then there was no error. To make the program complete, the corrected codeword should be sent to the user instead of the syndrome. The necessary additional steps are to find the column of H matching the syndrome, call it i, and then flip the i^{th} column of y. Probably the simplest way to accomplish this would be to exploit the correspondence between the column numbers of H and their representation as a binary number. That is, the bad column number is simply: $i = 4 \cdot s_1 + 2 \cdot s_2 + s_3$.

```
BEGIN
  LET vector = 'FORMAL u;
    BEGIN
      LET n = (u :length),
      LET ip ='FORMAL v;
        BEGIN
          LET s := 0;
          USING 1,,n FOR i DO s := s + ((u :index i) * (v :index i)) mod 2
        END';
      self
    END';

  LET  H_matrix = [self :vector [0,0,0,1,1,1,1],
                   self :vector [0,1,1,0,0,1,1],
                   self :vector [1,0,1,0,1,0,1]],

  LET   y = [1,1,1,1,0,1,1],

  LET syn = [H_matrix :index 1 :ip y,
             H_matrix :index 2 :ip y,
             H_matrix :index 3 :ip y];

  user :print syn
END
```

Figure 4: Partial Hamming Decoder

For purposes of demonstration, this trick is ignored and the example is completed by comparing the columns of H with the computed syndrome and then returning the index of the matching column. Since H is represented as a list of rows, the columns of H cannot be readily accessed. Without restructuring the program, a slightly roundabout method is proposed to find the matching column. The strategy is first to check the syndrome to see if it is all zeros. If so then no error occurred and y is sent to `user`. Otherwise the first row of `H_matrix` finds all the columns of its row vector that match the first element of the syndrome. This `vector` object then forwards these column numbers to row 2. Row 2 will then do the same except that it is comparing against the second element of the syndrome and will forward the remaining column numbers to row 3. Finally, the number that is emitted by row 3 is the value of i as described above.

The program to execute this algorithm is shown in Figure 5. A few subtle modifications have been made to the program. First, `vector` now takes two arguments, the list that is its vector, and its row number within the H matrix. Note that a fourth row has been added with the value of `self` which contains a reference to the environment object of the outermost block clause. The representation of syn has changed from a simple list to an object that internally contains a three element list which can respond to `:index` messages as do list objects and can also check to see if the syndrome consists entirely of zeros.

Providing that the syndrome is not zero, the first row vector is sent a `:filter` message to compare column 1 of the syndrome with column 1 of u. If they match then row 2 is sent a `:filter` message otherwise column 2 of row 1 is compared against column 1 of the syndrome. Messages are thereby filtered from row 1 through row 3 and any time a partial match is invalidated, then the row column number is incremented and row 1 is again sent the `:filter` message. When a match occurs at row 3, then the column number has been found and this number is sent to row 4 of the `H_matrix`. This causes a `:filter` message to be sent back to the outer block. The outer block has been embellished with its own `filter` method that simply interprets the number emitted by row 3 as i and flips the i^{th} bit via the `fix` method.

The program of Figure 5 demonstrates an interesting programming technique that can be used quite effectively in XCPL. Simply put, conditional clauses can often be replaced with a message send. The decision that would normally be computed by the conditional is instead handled by the interpretation of *selector* in the context of the value of *object*. To see this in action, recall that row 4 of `H_matrix` was defined as the value of the environment object for the outermost block clause. Thus the context of the value of the first three rows is that of `vector` objects whereas the fourth is the outermost block clause. By changing the context, the value associated with `filter` has changed and a different method is executed. Without this technique it would be necessary either to make a different `filter` method for row 3, in effect, a different `vector` object for row 3, or to include an additional `IF...THEN...ELSE` clause in the `filter` method to detect when the row number is 3 and the syndrome column matches the vector column.

```
BEGIN
  LET fix    = 'FORMAL vect; FORMAL i;
    BEGIN vect :set [i , vect :index i + 1 mod 2]; vect  END',

  LET  y = [1,1,1,1,0,1,1],

  LET filter = 'FORMAL syn; FORMAL i; self :fix [y,i]',

  LET vector = 'FORMAL u; FORMAL row;
    BEGIN
      LET filter = 'FORMAL syn; FORMAL i;
        IF (u :index i) = (syn :index row)
           THEN H_matrix :index (row+1) :filter [syn,i]
           ELSE H_matrix :index 1       :filter [syn,(i+1)]',

      LET ip ='FORMAL v;
        BEGIN
          LET s := 0;
          USING 1,,n FOR i DO s := s + ((u :index i)*(v :index i)) mod 2
        END',

      LET n = (u :length);
      self
    END';

  LET H_matrix = [self :vector [[0,0,0,1,1,1,1],1],
                  self :vector [[0,1,1,0,0,1,1],2],
                  self :vector [[1,0,1,0,1,0,1],3],
                  self ],

  LET syn = BEGIN
              LET a = H_matrix :index 1 :ip y,
              LET b = H_matrix :index 2 :ip y,
              LET c = H_matrix :index 3 :ip y;
              LET is_zero = '(a = 0) and (b = 0) and (c = 0)';
              LET index = 'FORMAL i;
                IF i = 1 THEN a ELSE IF i = 2 THEN b ELSE c';
              self
            END;

  user :print (IF (syn :is_zero) THEN y
                  ELSE H_matrix :index 1 :filter [syn,1])
END
```

Figure 5: Complete Program for (7,4) Hamming Decoder

3.1 The Eight Queens Example

The Eight Queens problem is one of the showcase problems often cited in computer programming. The task is to place eight queens on an 8x8 chessboard so that no queen is in jeopardy of capture. The rule for the queen chess piece is that it may capture any game piece lying along the same row, column, or either diagonal. The solution of this problem is a good application to try in XCPL since it addresses two issues that were not previously covered. First, there is the question of what should be the concurrent objects. In the Hamming decoder example, the choice followed naturally from the formulation of the vector operations. Secondly, whereas the first example had extremely regular message flow, this example involves *backtracking* when a chessboard configuration of queens is found to be unacceptable.

A solution (selected for clarity above all else) is shown in Figure 6. It follows from the capture rule that no two queens may be in the same row or column. Based on this observation, each column is an object and will contain exactly one queen. Each column moves its queen up and down to avoid capture from the other queens. When a capture is inescapable, backtracking is used to continue the search.

The chessboard is defined as a nine element list with the environment object of the outer block as the first element followed by eight queen column objects. The chessboard is created by sending `self` eight `:queen` messages within the context of a list. The `queen` method takes as a single parameter its index in the `board` list. Each `queen` object will define four operations on itself: `print`, `capture`, `move`, and `evade`, and then set its internal row numbers to 1. Note that the environment object for the outer block clause also has the `print`, `capture` and `evade` methods defined.

A first solution is found by sending the second queen column a `:move` message starting with row number 3. Providing the column number is not the last one, the `move` method will reset the row number and then send itself an `:evade` message to start searching for a valid row number for the *next* column starting with row 1. If the column number is 8, then a solution has been found and two `:print` messages are concurrently sent. The first is sent along with the row and column number as arguments to the `user` object. The other `:print` message is sent to the *previous* column object so that the entire chain of valid queen positions from column 1 through 8 will be sent to the `user` object. Notice that the succession of `:print` message terminates with the environment object of the outer block.

When the `evade` method object is created with a row number, it will send itself a `:capture` message to test whether capture is imminent for the projected new row number. If so, then the `evade` method is recursively invoked on the next row number until either `newrow` is off the board or a safe position is found. The `capture` method works by checking the received row and column number against the row and column number of the containing `queen` object. If the `queen` object can capture the projected coordinates then a value of `true` is returned, otherwise the `:capture` message is forwarded to to the previous column. This will continue until either a capture occurs or the `:capture` message reaches the outer block. The `capture` method for the outer block will always return a value of `false`.

After a safe position is found, the `queen` object for the next column is sent a `:move` message to assign the safe row number and continue the computation for the next column. If `newrow` is off the board, then capture is inescapable and the row number of the searching `queen` needs to be altered. This is accomplished by sending the *previous* column an `:evade` message which will continue the search for a valid row number starting with the row number one greater than the current row number. Since the `evade` method of the previous column

```
BEGIN
  LET evade   = 'user :print "NO MORE SOLUTIONS"',

  LET capture = 'FORMAL c; FORMAL r; false',

  LET print   = 'user :print :newline',

  LET queen = 'FORMAL col;
    BEGIN
      LET print = 'BEGIN
                     user :print [row,col] ,
                     (board :index col :print)
                   END',

      LET capture = 'FORMAL c; FORMAL r;
        IF (r = row) OR ((c - col) = (r - row :abs))
           THEN true ELSE board :index col :capture [c,r]',

      LET move = 'FORMAL r;
        BEGIN
           row := r;
           IF col < 8 THEN self :evade 1 ELSE (self :print)
        END',

      LET evade = 'FORMAL newrow;
        IF newrow <= 8
           THEN IF self :capture [col+1,newrow]
                   THEN self  :evade (newrow + 1)
                   ELSE board :index (col + 2) :move newrow
           ELSE board :index col :evade (row + 1)',

      LET row := 1;  % declare row as an integer value
      self
    END';

% main program
  LET board = [self,self :queen 1, self :queen 2,
                    self :queen 3, self :queen 4,
                    self :queen 5, self :queen 6,
                    self :queen 7, self :queen 8];

  board :index 3 :move 3
END
```

Figure 6: Eight Queens Program

may in turn cause an :evade message to be sent to *its* previous column, a backflow of :evade messages may occur culminating with an :evade message sent to the first element of the list board. An evade method is defined for the outer block so that if an :evade message reaches the outer block, it will report failure to find a solution.

Part of the beauty of this program is that from the first solution all subsequent solutions may be found. These additional solutions are extracted simply by sending column 8 :evade messages until the tough luck message from the outer block occurs. The unfortunate aspect of this program is that, for the most part, it is sequential in execution. The sequential behavior stems from the definitions of capture and evade. Capture performs a sequential search from right to left starting with the current column. This search could be done concurrently; however, the gain in performance would be negligible since the capture test is very simple and the number of positions to test concurrently grows only linearly with column number, peaking out at seven in column eight.

The evade method sequentially tests the new row numbers from 1 to 8 and then pursues the first one that is safe. The natural generalization is to start up a search not only on the *first* safe position but on *all* safe positions. With this approach the fixed chessboard arranged as a list of columns must be discarded in favor of a tree structure whose shape depends upon the progress made by each of the individual queen objects. This generalization of evade effectively changes the search from *depth-first* to *breadth-first* with two immediate ramifications. First it is no longer necessary to backtrack, thus :evade messages propagating from right to left will be removed. Secondly, the exact shape of the queen tree cannot be predicted beforehand.

A concurrent solution is shown in Figure 7. In accordance with the discussion above, the board list is gone and the move method has been subsumed as part of the queen method. The definition of queen now takes a second parameter called leftqueen in order to build the tree structure. Leftqueen is similar to "board :index col" of the previous solution in that it is always set to the value of the queen object of the column immediately to the left. Queen objects are generated on the fly whenever a safe row number has been found. The evade method has been replaced by a simple USING loop. For every safe row that is found, a queen with the safe coordinates is created with a value of leftqueen of the queen that found it. For the sequential version, evade recursed until a safe row was found or all the rows were tested, after which an :evade message was propagated backwards. The value computed by evade was determined by either the :move message to the *right* or the :evade message from the *left*. For the concurrent version, iterations generating a safe position yield a value of a message send to to the queen object, and unsafe positions yield a value of undefined or '???', which is the way Ω is written in program texts.

Initially the outer block starts up a queen object for each row with column number set to 1. Each instance of queen object will first define the print and capture methods on itself and then start testing the eight rows of the next column. When the final column is reached, the print method is invoked to recursively output the path from leaf to root, that is from the column eight queen to the column one queen, and then terminate with the send to the print method of the outer block.

```
BEGIN
  LET capture = 'FORMAL c; FORMAL r; false',

  LET print   = 'user :print :newline',

  LET queen = 'FORMAL col; FORMAL row; FORMAL leftqueen;
    BEGIN
      LET print = 'BEGIN
                     (leftqueen :print).
                     user       :print [row,col]
                   END',

      LET capture = 'FORMAL c; FORMAL r;
        IF (r = row) OR ((c - col) = (r - row :abs))
           THEN true
           ELSE leftqueen :capture [c,r]',

      USING 1,,8 FOR newrow DO
        IF self :capture [col + 1,newrow] THEN ???
           ELSE IF col < 7
                   THEN self :queen [col + 1,newrow,self]
                   ELSE (self :print)
    END';

  % main program
  USING 1,,8 FOR i DO self :queen [1,i,self]

END
```

Figure 7: Highly Concurrent Eight Queens Program

3.2 Gaussian Elimination

Numerical analysis is an important application domain for XCPL. A representative problem from this domain is to solve a set of n linear equations with n unknowns. The solution technique chosen is Gaussian elimination where each equation represents one row of a matrix. To solve for the unknowns, the matrix is first put into *reduced row-echelon form* and then *back substitution* is used to output the computed values for the unknowns. The row reduction phase will be solved for with *pivoting* included to minimize round off error. The algorithm consists of the following steps with j ranging from 1 to n:

1. Find pivot row for column j.
2. Move pivot row to the top of the matrix.
3. Adjust pivot row so that value of column j is 1.
4. Add multiples of the top row to the rows beneath it so that column j is zero.
5. Cover top row, increment j and repeat.

If arithmetic operations are considered the fundamental steps, then it immediately follows that this algorithm requires $O(n^3)$ steps, where n is the number of unknowns. If the rows are organized as objects as was done in the Hamming decoder, then step 4 can be performed concurrently for each of the n rows. Thus an $O(n)$ factor is performed concurrently reducing the number of sequential steps to completion to $O(n^2)$. Figure 8 shows the program for this algorithm. The row objects are created by the `matrix_row` definition and are instantiated inside the list named `matrix`. By indexing into `matrix`, the rows are able to send messages to each other.

For the program in Figure 8, `first` references the top row of the matrix. For step 2, rather than move the pivot row to the top, the same effect can be achieved by simply sending the pivot row a `:cover` message. The logical variable `covered` is used to keep track of which rows have been covered by step 5 and are no longer involved in the computation.

The `find_pivot` method performs step 1 by scanning the uncovered rows for the row with the largest absolute value for column j. When the outer block is sent a `:find_pivot` message then all rows have been tested so a `:cover` message is sent to the largest row. The `cover` method performs step 3 and then sends a `:mpy` message to `first` providing the last column has been reached. The `mpy` method performs step 4 and also forwards the `:mpy` message to the next row. When the outer block receives a `:mpy` message, then the multiply phase for column j is guaranteed to be safely underway and the find pivot operation for column $j+1$ can be safely started.

```
BEGIN
  LET mpy = 'FORMAL j; FORMAL u; first :find_pivot  [0,j+1,0.0]',

  LET find_pivot = 'FORMAL maxrow; FORMAL j; FORMAL maxval;
    matrix :index (maxrow + 1) :cover j',

  LET matrix_row = 'FORMAL v; FORMAL rownum;
    BEGIN
      LET find_pivot = 'FORMAL maxrow; FORMAL j; FORMAL maxval;
        matrix :index (rownum + 2) :find_pivot
         (IF covered THEN [maxrow, j, maxval]
             ELSE IF maxrow = 0 THEN [rownum, j, (v :index j :abs)]
                     ELSE IF maxval > (v :index j :abs)
                             THEN [maxrow, j, maxval]
                             ELSE [rownum, j, (v :index j :abs)])',

      LET mpy := 'FORMAL j; FORMAL u;
        IF covered THEN matrix :index (rownum + 2) :mpy [j,u]
           ELSE BEGIN
                  LET m = v :index j; v :set [j,0.0];
                  USING j + 1,,n FOR i
                    DO v :set [i,v :index i - (m * (u :index i))],
                  matrix :index (rownum + 2) :mpy [j,u]
                END',

       LET cover = 'FORMAL j;
         BEGIN
           covered := true, LET m = v :index j; v :set [j,1.0];
           USING j + 1,,n FOR i DO v :set [i, v :index i / m].
           user :print [v];
           IF j < (n - 1) THEN first :mpy [j,v] ELSE user  :print :newline
         END',

       LET n = (v :length),
       LET covered := false;
       self
     END';

  LET matrix = [self,self :matrix_row [[3.0, 2.0,-1.0, 1.0],1],
                     self :matrix_row [[6.0, 6.0, 2.0,12.0],2],
                     self :matrix_row [[3.0,-2.0, 1.0,11.0],3],
                     self :matrix_row [[1.0, 1.0, 1.0, 1.0],4],self];

  LET first = matrix :index 2;
  first :find_pivot [0,1,0.0]
END
```

Figure 8: Gaussian Elimination Program

3.3 Generating Primes by Sieving

The problem addressed in this example is to generate prime numbers by constructing a sieve that filters out all the non-prime numbers. The technique used is one devoid of tantalizing number-theoretic tricks but elegantly uses the concurrent objects of XCPL. The input to the sieve is a stream of integers which contains all the prime numbers and has the property that the integers are emitted in strictly increasing order. The simplest such stream would be the natural numbers. The sieve is made up of a linear chain of objects where each object holds one prime number. Each sieve object receives potential prime numbers to test for divisibility. If the test number does not evenly divide, then it is sent to the *next* sieve object of the chain. If the test number reaches the end of the sieve, then it must be a prime and is made into a sieve object at the end of the chain.

As was mentioned previously, the number generator could simply emit the natural numbers, but this would be wasteful since all even natural numbers aside from 2 are not prime. A first improvement therefore would be to omit multiples of two, that is, to emit only the odd numbers. Further improvements would be to omit multiples of $2, 3, 5, 7, 11$, *etc.* A number generator of this type is called a "wheel" [Guy]. The wheel is composed of an "addendum" and "spokes". The addendum is the product of the prime numbers from 1 up to some finite limit. The spokes are the integers relatively prime to the addendum. The wheel operates by maintaining an accumulator which is a multiple of the addendum. The accumlator is added to each spoke and then sent to the sieve for primality testing. After each spoke has sent a number to the sieve, the accumulator is advanced to the next multiple of the addendum.

For example, consider the addendum: $1 \cdot 2 \cdot 3 = 6$. This addendum would result in a two spoke wheel of 1 and 5 that would generate the sequence: $0+1, 0+5, 6+1, 6+5, 12+1, 12+5, 18+1, 18+5, \ldots = 1, 5, 7, 11, 13, 17, 19, 23\ldots$. Every integer in this sequence is relatively prime to 2 and 3, and the first non-prime generated is 25. The number of spokes for a wheel is determined by calculating Euler's Function ($\phi(addendum)$) for the addendum. The wheel used in the program of Figure 9 consists of 8 spokes with values of $1, 7, 11, 13, 17, 19, 23$, and 29, and has an addendum of $2 \cdot 3 \cdot 5 = 30$. The objects generated by this program are shown graphically in Figure 10. In this graph, circles represent objects and the edges represent the message flow between objects. The wheel in Figure 10 corresponds to the list `wheel` of the program. This list consists of 9 objects, 8 `spoke` objects and an `idler` object. The `spoke` objects are initialized with the offsets: $1, 7, 11, 13, 17, 19, 23$ and 29, respectively. When a `spoke` object receives an `:add` message, it will add its offset value to `k`, which is the current multiple of the addendum. This sum is then sent to the sieve via `hub` to check and see if the new sum is prime. The `add` method will also concurrently forward `k` to the next spoke. The ninth element of the wheel list is the `idler` object, when it receives a message from the 8th spoke, `idler` will add the addendum to `k`, thus advancing to the next multiple of the addendum.

The `check` method of the `sieve` objects works by checking if the number received is evenly divisible by the prime number of the `sieve` object. If it does not evenly divide and a next prime exists, then the the number received is forwarded to the next prime. If no next prime exists, then the number received is prime and is stored by creating a new `sieve` object. Whenever a `sieve` object is created, the value of the prime is sent to the `user` object as the argument to the `:print` message.

Unlike the previous examples, this program derives its concurrent behavior from operating in a pipelined fashion. Since the `add` method of `spoke` concurrently sends to both

```
BEGIN
  LET spoke = 'FORMAL v; FORMAL i;
    BEGIN
      LET add = 'FORMAL k;
        BEGIN
           hub :check (v+k),
           wheel :index (i + 1) :add k
        END';
      self
    END',

  LET sieve = 'FORMAL v;
    BEGIN
      user :print v;
      LET check = 'FORMAL p;
        IF p mod v = 0 THEN []
           ELSE IF next = ??? THEN next := self :sieve p
                                 ELSE next :check p';
      LET next := ???;
      self
    END',

   LET idler = 'FORMAL delta;
     BEGIN
       LET add = 'FORMAL k;  wheel :index 1 :add (delta+k)';
       self
     END';

   LET hub = self :sieve 7;

   LET wheel = [self :spoke [ 1,1], self :spoke [ 7,2],
                self :spoke [11,3], self :spoke [13,4],
                self :spoke [17,5], self :spoke [19,6],
                self :spoke [23,7], self :spoke [29,8],
                self :idler 30];

  wheel :index 3 :add 0
END
```

Figure 9: Prime Generator Using an 8-Spoke Wheel

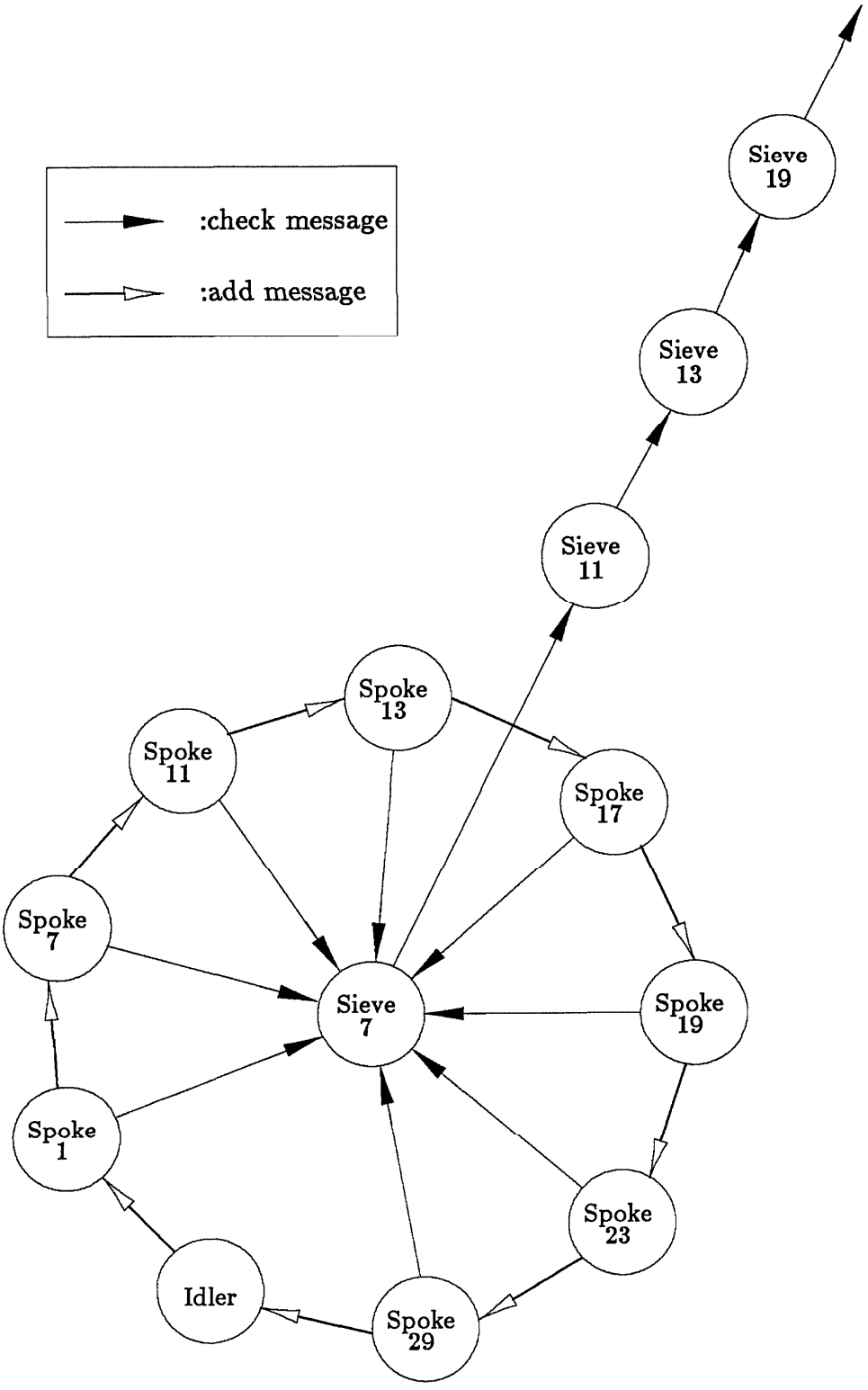

Figure 10: Object Graph for Primes Program

hub and the next spoke object, many potential prime numbers may be in in the sieve (see Figure 10) at any given instance. Thus both the tests for primality as well as the generation of potential prime numbers proceeds concurrently. A drawback to this strategy is that the `sieve` objects near the beginning of the sieve will be executed far more often than the latter stages. This problem was addressed in part by using the wheel which substantially reduces the number of test cases. A better solution that would completely solve the uneven load problem would be to make the size of the wheel grow dynamically by adding more spokes and appropriately increasing the size of the integer stored in the `idler` object.

I write to discover what I think.

- Daniel Boorstin

4 Research Overview

The overall goal of the XCPL experiment is to provide a semantic bridge between object programming and ensemble machines. Object programming evolved as a method to manage programming complexity by data encapsulation and hierarchy, while ensemble machines evolved from an understanding of the physics governing the scaling properties of VLSI circuits. Although their lineages are quite diverse, object programming and ensemble machines can be viewed as a natural confluence for the development of future computer systems. An example of this confluence is the issue of locality. In object programming, the computational methods that can be performed on an object are made part of the object. The names used to denote computational methods have meaning only in the receiving objects. From the viewpoint of a sending object, the name may have multiple meanings, a different meaning for each possible destination object. In contrast, function names as found in languages such as ALGOL denote exactly one definition from the viewpoint of both sender and receiver. Method names can therefore be said to be more "powerful". For ensemble machines, the mechanisms that perform operations (methods) are physically placed next to the mechanisms that hold the state information (object data) for the computation. This organization is advantageous because less space, time, and energy are necessary to drive a signal on a short wire than on a long wire.

A second example of the confluence between object programming and ensemble machines is the exclusive use of message passing to exchange information. The alternate scheme for exchanging information would be to use storage that could be shared by more than one object. For object programming, restricting external accessibility of object variables protects the variables from corruption by other, possibly malicious, rogue objects. Instead, the object can only be corrupted by its own methods. Ensemble machines have no shared storage since by definition, an ensemble machine is a collection of computing elements that are interconnected by a communication network.

To develop the confluence between object programming and ensemble machines, a new computational model is needed that captures the desirable properties of both. The primary properties of this model are the following:

- Objects create other objects (new).
- Objects change the behavior of other objects only by message passing (send).
- Objects contain persistent data (history-sensitive).
- Objects exercise discretion in receiving messages (valor [1]).
- The only prerequisite for sending a message is that the sending object know the address of the destination object (locality).
- Messages from a *single* source sent to a single destination arrive in the order sent (message order preservation).

[1] "The better part of valour is discretion." — Henry IV, Part I, V, *iv.*

- Messages from *multiple* sources arrive at a single destination in arbitrary order (arbitrary delay).

This new computational model was the basis for the programming examples of Section 3. In realizing the computational model there are difficult problems to be resolved. The semantics of object programs must concur with the new computational model, and ensemble machines must be suitably enhanced to support the new model. Object programming languages have evolved solely on uniprocessors, and from this background have developed a chronic dependence upon sequential execution. Although the state variables of the various objects used in a computation are kept separate, the programming model requires messages to be executed one at a time. For the case where a message send will cause a second send and so forth, the pending message sends are suspended in a last in first out (LIFO) fashion. Sequential execution in conjunction with the LIFO suspension rule is used to perfectly synchronize every step within a program. From an implementation viewpoint, this scheme for message passing closely resembles the activation chains used in subroutine calling in languages such as ALGOL. Such a message passing scheme is fundamentally different from that of the XCPL computational model where message passing is both autonomous and concurrent. The programmer cannot be oblivious to the differences for they are fundamental to XCPL object interaction.

To take advantage of the concurrent nature of ensemble machines, concurrent behavior was made fundamental to the XCPL computational model. In turn, this decision mandated a substantial change in the semantics of objects, hence XCPL was defined to express computations based on the new computational model. At the other extreme, XCPL, as well as sequential object languages, has capabilities that include the creation of new objects and message passing based solely on *reference*. These capabilities are included as properties of the XCPL computational model, and are not intrinsic to ensemble machines. Management of object creation and deletion and message routing must be hosted on ensemble machines by providing a suitable runtime environment.

Proceeding from these general remarks, the XCPL experiment can be viewed as both a software and a modelling experiment. From the software perspective, the problem is to devise a programming language which implicitly incorporates the weak synchronization and concurrent properties of the computational model, while still retaining intelligibility as a programming notation. Programming strategies for accomplishing these goals have been demonstrated in the examples contained in the preceding section. Viewed as a modelling experiment, the problem is to first treat each environment object as if it were a separate machine and then to *optimize* the internal operation of each machine to minimize time and space requirements. Results of this optimization are used in conjunction with a static analysis of the communication patterns between objects to provide a basis for deciding how each object should be realized. For example, whether to dedicate a processing node to an object or to assign multiple objects per node. If the latter choice is made then questions arise as to how the object should be allocated, from fixed storage, heap space, a stack, *etc.* The goal of this analysis is to anticipate the dynamic nature of the XCPL program so that the resources of the ensemble are not overly taxed. This goal does not imply that all runtime process management is eliminated, but rather that it is simplified as much as the program text will allow.

The key to success for both facets of the experiment ultimately depends upon the sophistication of compilation. The XCPL experiment does not use a single machine model that is then hosted on an actual processor, but rather uses a progression of machine models,

each of which is referred to as an *engine*. Each new engine directly corresponds to an added layer of refinement in the compiler. As the compiler progresses in sophistication, the engines become more exact in nature. Furthermore, the runtime systems used to orchestrate the computations are provided with better scheduling information so less effort is needed to manage the objects at runtime.

Whereas any matched set of a compiler and engine constitute an XCPL programming system, the goal of the progression is to develop an engine that closely resembles coarse grain ensemble machines such as the Caltech Cosmic Cube [Seitz85] or Intel iPSC [Intel]. Emphasis therefore is shifted away from heavy reliance on the runtime system to extensive flow analysis. The later stages of the compilation process must identify the concurrent components of an XCPL program, extrapolate them into an executable process structure, and then map the process structure onto the ensemble. The last step in its most general form is an *NP*-complete problem. An exact solution would have to take into account such diverse information as the shape of the process structure, topology of the ensemble, and size of the nodes of the ensemble. The final step is further complicated by permitting the process structure to evolve during a computation. Therefore, finding the optimal solution, providing it could be defined, is a hopeless task. The XCPL compiler instead tries to find an embedding which does a reasonable job of load balancing, with respect to both message traffic and node utilization. Information used for load balancing is extracted at compile time by attempting to characterize object behavior using a number of parameters. The success of this phase of the compilation process depends both upon the success of the compiler and also upon the amenability of the ensemble. Analysis has shown that richly interconnected ensembles such as binary n-cubes perform well for all but the most malevolent embeddings [Steele].

4.1 Compiler and Engine Progression

The first XCPL engine and compiler have been implemented and used to wring out the examples of Section 3. The compiler is recursive descent and requires two passes. To understand the rôle of the compiler, an explanation of the representation used for environment objects is necessary. Each environment is described by a template called an "environment descriptor" that contains all the necessary information to build the environment object at runtime. An environment descriptor is broken down into the following sections:

- Reference to parent link
- Reference to caller link
- Declared variable table
- Temporary variable table
- Control flow graph

The parent reference is established at compile time and refers to the object that lexically contains the current object. The caller reference is possibly established at runtime and refers to the object that is requesting a service of the current object. Every variable defined by either a `LET` or `FORMAL` declaration is inserted into the declared variable table. For assignment, keyword, and send clauses (see Figure 1) the compiler must generate instruction sequences to accomplish the action associated with each clause. Many translations of clauses

require the use of temporary variables to store intermediate results. These variables are purely local to the object and are allocated from the temporary variable table.

The kernel of instructions used by the compiler is called *ecode*. The ecode kernel is common to all the engines and is the starting point for all compiler optimizations. Every environment descriptor contains a control flow graph represented by a Series-Parallel (S-P) directed graph [Harrison] whose vertices are ecode instructions. Two types of edges are used in the S-P graphs. Data dependence edges denote that the value *defined* by the vertex at the tail of the edge is *used* by the vertex at the head of the edge. Control dependence edges denote that the vertex at the tail must be executed before the vertex at the head.

The ecode vertices may have multiple input and output edges. When a value is produced by a vertex, either all the output edges will receive the value or just one (mutual exclusion). The same is also true for the input edges. An example of this behavior is the ecode **if** instruction which uses a single value but produces two mutually exclusive outputs. These two outputs can be either data or control dependence edges. The compiler decides whether two vertices should be connected in a series or parallel fashion by examining the combinator used to compose the original clauses and by recognizing the data dependencies within a clause, *e.g.* an extended send clause.

For the first engine, every block and method definition of an XCPL program is treated as an independent environment object. On the first pass all variable names encountered are considered to be local to the object where they are found. Proceeding from this simplification, the tasks of the first pass are to build the environment descriptors and generate ecode instruction sequences for each of the syntactic clauses. The environment descriptors are stored away in a table called the Environment Descriptor Table (EDT). The second pass scans the code and the EDT to check whether the assumption about name locality was true or not. For every instance where the assumption was incorrect, additional ecode instruction sequences are inserted to effect the inter-object access request.

The first engine exercises only the rudimentary capabilities of the compiler and thus relies heavily on the runtime system to schedule both instruction and object execution, and to recognize object types for the appropriate message dispatch. The instructions within a control flow graph are only partially ordered based upon both specified and detected data and control dependencies. Data dependencies are often generated by the use of reservations to allow computations to continue when the reply values are not yet available. The engine executing the program therefore has to check to see if the values necessary for an instruction are available before executing the instruction. If not, no harm is done except that the resources of the engine were wasted. By inferring more data dependencies, the control flow graphs can be streamlined to reduce the number of non-productive attempts to execute instructions. This problem was solved in part by the first pass of the compiler which detects dependencies within a clause but relies on the programmer for stating all additional dependencies between clauses. By constructing def-use chains [Kennedy], the compiler can detect additional dependencies between clauses.

To further improve runtime performance, in particular the task of process placement, program flow analysis can be used to construct an initial prediction of the concurrent object formation. The prediction is then used to find an initial mapping or embedding of concurrent objects to processing sites within the ensemble. An immediate ramification to this approach is that a "closed world" assumption must be made about the XCPL program under scrutiny. In practice this means that all the definitions for objects must be available to the compiler so that the "acquaintances" of the objects, in particular message selectors and environment object types, can be inferred. Programs are not allowed to introduce new object types nor

message selectors at runtime. The flow analysis is separated into two inter-related tasks: data type analysis and control flow analysis.

As was mentioned in Section 3, decisions in a program can often be accomplished as the result of a message dispatch rather than a conditional clause. For such cases the destination object type may not be uniquely determined but possibly limited to a set of object types. Performing type analysis on the variables used within an object will attempt to restrict the range of possible destination object types. Since the type analysis applies equally to both primitive and environment objects, determining the object type at compile time can be used advantageously in two ways. If the destination object is determined to be a primitive data type, then the actual message send can be replaced by a special instruction that performs the primitive operation in place. If the destination object is an environment object, then the type information can be saved and used later for determining the object graph for the program.

The collecting of type information is effectively the inclusion of context-sensitive information in a generally context-free language. With regards to the compilation process, gathering type information can be done at three distinct layers:

1. Entirely within a clause.

2. Between clauses entirely within an object.

3. Between objects.

Accruing type information inside a clause is particularly simple for it can be done in conjunction with the first pass of the recursive-descent compilation. Rather than using the recursion to determine only whether a clause was recognizable, the return from the recursion is expanded to include the type of the clause. (Note that in some cases the type is not necessarily unique, for example an "`IF..THEN..ELSE`" clause.) The S-Algol compiler [Davie] performs a similar manœuver but its goal is to *enforce* a set of typing rules, whereas the XCPL compiler merely wants to accumulate such information.

For the case where the data type is not unique, the type of a clause may be expressed as an equation. By considering all the equations within an object, some or all of the equations may be *solved* so that the data type is uniquely determined for some variables. Iterative algorithms that solve the equations by relaxation have been used for similar type analysis problems [Holstege]. The type analysis described so far covers variables declared by `LET` clauses; variables declared by `FORMAL` clauses cannot be typed using only local information. To include such variables in the type analysis, it is necessary to examine the interaction between environment objects. Fortunately the type information gathered so far provides an initial relation of the interacting object types. The object types of formal parameters may then be explored. By determining the type of formal parameters, more type information is then available about variables internal to an object. This in turn may provide more type information about the interaction between object types. Thus a feedback loop is established between layers 2 and 3 of the type analysis. The number of iterations actually performed between the two layers is a matter of trading-off compile time for execution time.

Whereas type analysis investigates the type relationship between objects, control flow analysis investigates the *phase* relationships between objects. Examples of phasing relations include whether a collection of objects are mutually exclusive, concurrent, strictly sequential or pipelined sequential. Phase information is produced by examining the object interactions found in the type analysis in conjunction with the control flow graphs.

The typing and phasing information gathered by the above techniques represents an analytic behemoth. Fortunately the goal remains steadfast at finding an initial object graph that can be loaded onto the ensemble. The task is governed by a need to match the grain size of each processing site and also to utilize all processing sites dedicated to the computation. The environment descriptors are tree structured by the parent reference link. The strategy offered is simply to collapse sequential objects bottom up until the grain size is reached and to dispatch concurrent objects into different processing sites top down until all processing sites have been put to work.

Whereas the strategy is simple, the difficulty lies in determining the initial shape of the object graph. The three general cases for the shape analysis are:

1. The object graph is static.

2. The object graph is recursive and regular.

3. The object graph is spaghetti.

The third case corresponds to joint failure of the flow analysis to find the concurrent structure and of the programmer to convey the concurrent formulation to the compiler. For such cases the runtime system must be relied on for all object placement and allocation. The "real estate" process planned for the Cosmic Cube is an example of such a runtime system. The overall strategy for the real estate process is to query neighboring processors to find the best place to locate a new process.

Case 1 corresponds to where not only the type of the concurrent objects is constant, but also the values are constant. Examples of such programs are the Hamming decoder, Gaussian elimination, and sequential Eight Queens programs of Section 3. Case 2 is motivated by research in analyzing LISP data structures [Muchnick]. To summarize, the technique is to combine the data type equations and S-P flow graphs into data flow equations, and then translate the new equations into tree grammars. The result is that the shape of the object graph is expressed as a tree grammar, and by exercising the grammar, an object graph is generated. Examples of programs that result in tree grammars are the concurrent Eight Queens program, whose shape can be bounded by an 8-tree of depth eight, and the primes sieve, with shapes of a constant ring and an infinite chain.

In conclusion, a strategy for the flow analysis has been proposed and the overall approach outlined. The compiler and engine experiments to be conducted next are summarized as the following:

1. Use SP-graphs for sequencing ecode.

2. Gather type information during recursive descent parsing, new instructions for handling primitive operations.

3. Propagate type information via relaxation algorithm.

4. Inter-object type analysis.

5. Collapse objects to meet grain size.

6. Convert data flow equations into tree grammars.

7. Execute tree grammars to generate an object graph.

4.1.1 Ecode Instruction Kernel

The ecode instruction kernel consists of eight operations that are considered to be fundamental. These instructions will always be supported and any instructions added later will always be of the "less powerful" but in some sense "less expensive" variety. Research in actor semantics has isolated the three primitive operations of "new", "send" and "become" as fundamental to actor computations [Agha]. The ecode kernel contains both a new and send instruction and the remaining six instructions could easily be interpreted as the components of a become operation. The eight instructions are described as follows:

T_k := **new** *edi*
: Creates a new instance of the object whose environment descriptor is contained at index *edi* in the EDT. The reference value of the new object is assigned to T_k.

T_k := **send** *obj sel arg*
: The message passing primitive of ecode. *Obj* must evaluate to an object, *sel* to a symbol and *arg* can be any value. The effect of the send is to build a message from the **send** instruction and enqueue the message. The value of T_k is left pending until a value is returned from *obj*. Note that execution can continue while T_k is pending, execution will wait for the reply value only when the value of T_k is demanded.

reply T_k
: The counterpart to **send**, the **reply** instruction sends a reply message with the value of T_k to the object whose value is that of the caller link in the environment descriptor.

T_k := **list** *n*
: Allocate a list of length *n* and store its value reference in T_k. In XCPL lists are predefined but not primitive objects. Two special instructions are dedicated to declaring lists.

element *i l val*
: Store *val* in the list *l* at index *i*. This instruction is used primarily in creating new lists. Programmers must use the more cumbersome `index` and `set` methods to access list elements.

move *src dest*
: Assign the value stored in *src* into the variable *dest*. Note that both *src* and *dest* must be variables local to the object.

if *pred*
: The conditional instruction of XCPL, *pred* must be a Boolean value, if *pred* is true then the `true` output edge is followed, otherwise the `false` edge is pursued.

demand T_k
: Normally execution continues until progress cannot be made without a result. The demand instruction inhibits the object from continuing until the value T_k is available.

Although this list summarizes the function of each of the different instructions, their operation might be better understood by considering them in conjunction with the first XCPL engine whose block diagram is shown in Figure 11. The major components of the engine are the Task Processor, Message Dispatch and various data type handlers. To keep

the first engine as simple as possible, all runnable objects share a common task queue and all messages in transit share a common message queue. A round robin scheduler is used to interleave the processing of tasks and messages. A further simplification is that the control flow graph is represented as a code string rather than an S-P graph, thereby imposing a predefined, total ordering on instruction activation within an object.

Messages are fetched from the Message Queue by Message Dispatch that examines the type of the destination. If the destination type is primitive, then one of the primitive data type handlers is invoked to process the message. If the destination type is user defined then the message is passed to Object Handler which first checks to see if the message selector is one of the reserved symbols of `:set`, `:get`, or `:value`. As was mentioned previously, for the first engine all environment objects are considered independent objects, therefore every variable access across an environment boundary, but within lexical scope, must be accomplished by message passing. The `get` method is used to get the value of a distant variable and the `set` method is used to assign the value of a distant variable. The `value` method is used to initially enqueue an object into the Task Queue. Note that when an object is created with the **new** instruction, it is not immediately placed in the Task Queue, rather it waits for a `:value` message before enqueueing itself.

If the message selector does not match one of the three special selectors of above then it is presumed to be a user defined symbol. The variable table of the environment descriptor for the destination is searched for the symbol. If the symbol is not found then a programming error has occurred and an appropriate error message is sent to the object `user`. If the symbol is present but not defined to be a method, then the message is simply requeued in the Message Queue, thereby deferring processing of the message until the object is ready. If the symbol is bound to a method, then the index for the environment descriptor of the method is passed to Object Maker which will do the actual instantiation of the object. Object Maker will also fill in the caller link with the sender reference of the message and then enqueue the method object onto the Task Queue.

The block diagram of Figure 11 succinctly describes the interactions between the different components of the first engine. For example, note that only the **new** instruction of the Task Processor and method of the Object Handler can invoke the Object Maker. Note also that only the **reply** instruction of the Task Processor does not requeue the object onto the Task Queue. Completion of a computation is detected when both the Task and Message Queues become empty.

4.2 Relation To Other Work

XCPL is the culmination of several relatively disjoint approaches to concurrent computing. XCPL is strongly rooted in the Cosmic Cube/Cosmic Kernel experiment [CPGCC], especially with regards to the computational model. The expressive power of XCPL was heavily influenced by Hewitt's actor systems [Clinger]. Both of these systems are compared to XCPL, as well as the programming languages OCCAM and Smalltalk. The intent here is not to discuss the similarities, that are for the most part quite obvious, but to focus on the differences, which are often quite subtle. Note that this list of examples is not exhaustive but rather representative of the different approaches to computing in a concurrent environment.

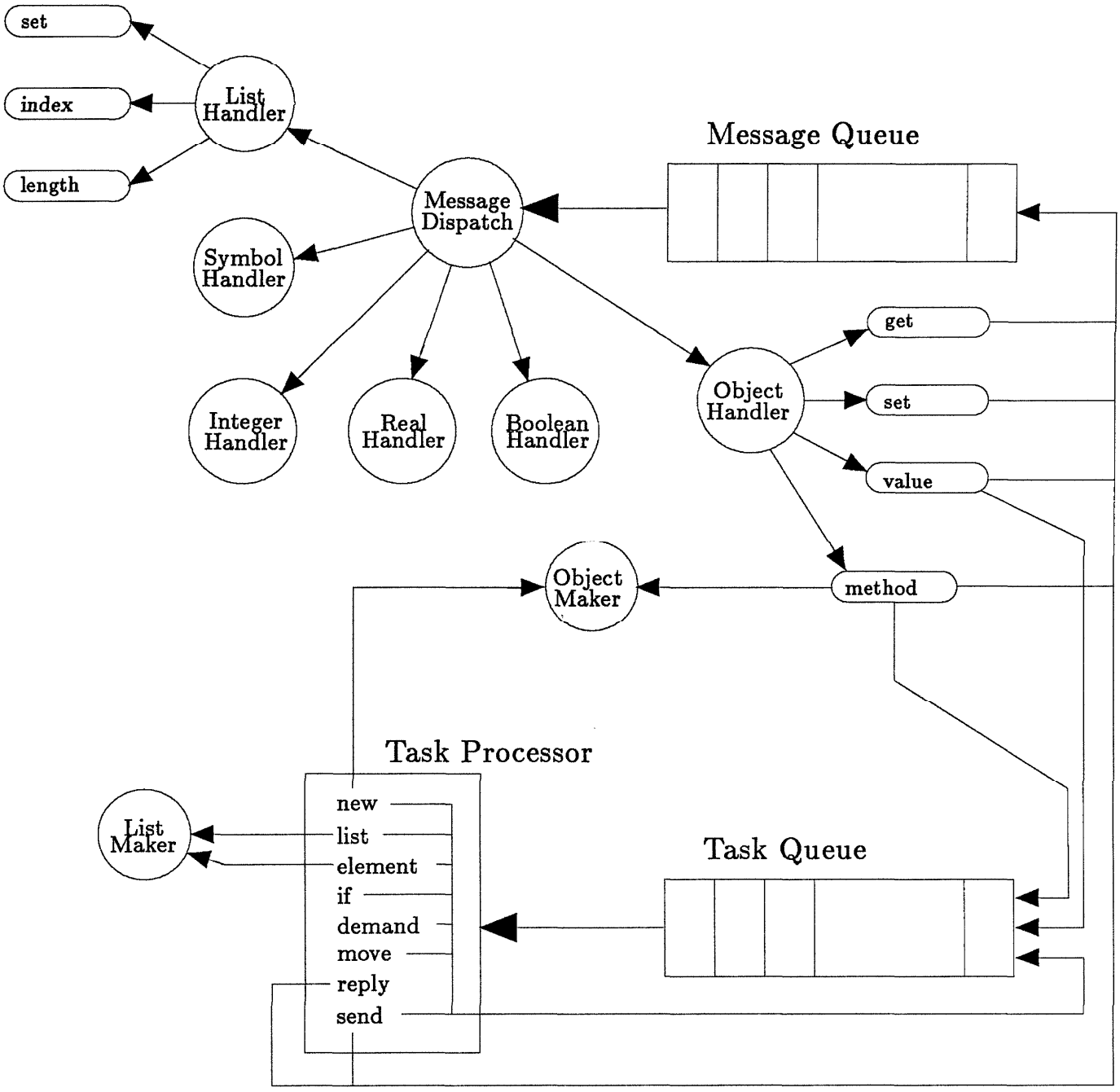

Figure 11: Block Diagram of First XCPL Engine

4.2.1 Cosmic Cube

Cosmic Cube is the archetype ensemble machine and Cosmic Kernel is its resident operating system. XCPL uses the same computational model as the Cosmic Cube experiment and the ecode kernel was designed under the impetus of the Cosmic Kernel. The Cosmic Cube supports a message passing environment based on a multiple process per processor model. Each process is an instance of a *sequential* program that interacts with other processes only by message passing. Send and receive message operations are implemented as system calls that are integral to the Cosmic Kernel operating system. These system calls are executed from running processes and are best thought of as creatings coroutines that perform the desired message action and then have to be later synchronized with in order to detect completion of the message action. In this context, completion means that a complete message has arrived for the case of a receive operation, or for the case of a send operation, that the message has completely left the environment of the sending process.

By far the most important aspect of this message passing system is that the only prerequisite for one process to send a message to another process is that the sender have the *address* of the receiver. The source of this address may have been determined at compile time, or may have been the content of a message received by the sending process. The only restriction placed on process interaction therefore, is that such interaction obey the actor law of locality [Baker].

The operating system is designed in two layers, the inner layer provides the message passing primitives that are invoked by system calls and the outer layer contains a set of special, privileged processes that are accessed only by message passing. To support the creation of processes on the fly, the outer layer contains a process called the "spawner". This process responds to requests from other processes to create a new process or dispose of a current one, providing in both cases the process to be acted upon resides on the same node as the spawner process.

Cosmic Kernel was a deliberate compromise to develop a set of low-level message passing primitives that were "usefully" abstracted from the hardware structure of its host ensemble machine. All operations related to the ensemble, namely message passing and the creation and destruction of processes, have to be explicitly included as operations outside the domain of the hosting language. Given that the send and receive operations behave as coroutines, the necessary explicit synchronization can be tedious. Furthermore, since every process is a separate program that has to be loaded into the ensemble, a process structure for the ensemble is not lexically contained within one program, at best it can be contained within one directory.

From this interpretation of the Cosmic Cube experiment, the XCPL experiment is a deliberate compromise to develop a language framework that is "carefully" abstracted from the message passing primitives developed in the the Cosmic Cube experiment. The compromise for XCPL is between object programming and the message system properties of ensemble machines. This amalgamation is clearly evident in the retention of the Cosmic Cube's send primitive as the **send** ecode instruction. The behavior of the send primitive permeates XCPL, except that the coroutining and subsequent synchronization have been effectively handled by the compiler. In compiling XCPL syntactic clauses into Cosmic Cube message primitives, send primitives are generated for the send clauses and also whenever it is necessary to access a variable across a process boundary.

The receive primitives of Cosmic Kernel are completely absorbed by the activation of method environments and by the built-in capabilities of block environments to react to

messages. The matching of a selector symbol to a method name can easily be accomplished in Cosmic Kernel by initiating a receive operation for every possible type of message and then waiting for one of the messages to arrive. The matching could also be done by using the "probe" operation which gives programmers the capability to test whether a message is present before having to commit to receiving it. For the probe operation, the different messages types would be continually probed for until a message is present, the corresponding receive operation would then be executed.

4.2.2 Actor Languages

XCPL is an actor language, though it is unique in that it was derived as a generalization to the semantics of ALGOL rather than LISP. The most commonly used actors, called *primitive serializers*, correspond to a high degree to the concurrent objects of XCPL. The primary difference is that a primitive serializer has only a single input queue whereas an XCPL object manages multiple input queues by manipulating the variable name table for the object. By modifying the name space, an XCPL object may discriminate on which queues it accepts messages from. For example, if an environment object has a variable not bound to a method and a message is received whose selector matches the variable name, then the message will not be received until the variable becomes bound to a method definition.

The computational model used by actor systems and the model used by XCPL and the Cosmic Cube experiment exhibit a curious symmetry regarding message order. As was mentioned previously, an XCPL object can change the arrival order of messages by adjusting the bindings of names to methods inside the object. Actor systems do not have this capability and messages must be received strictly in the order they arrive. If an incoming message is to be singled out before accepting any others, an actor must internally queue all the intervening messages. The XCPL model considers queuing to be inherent in the message passing system.

From the viewpoint of a sending object, the XCPL message system guarantees the sender that messages destined for a single common object will arrive in the order they were sent. In other words, message order is preserved on sending. The actor model does not require that message order be preserved between actors in direct communication, therefore unless the sender and receiver agree on additional protocols, messages will arrive at the receiver in arbitrary order. The advantages of one model over the other are clearly debatable, since examples demonstrating the superiority of either can be readily constructed and are inconclusive. Our practical experience has been that the preservation of message order simplifies programming, especially when considering the "side-effects" of environment objects processing messages.

Both XCPL and the actor languages provide for the creation of new objects (actors) at runtime. The only appreciable difference between the two is that actor languages allow actor definitions to be either modified or created at runtime. As was mentioned previously, in order to accomplish program flow analysis, the XCPL compiler requires that all definitions and all syntactic names be present at compile time. Therefore XCPL does not permit names and definitions to be introdcued on the fly.

4.2.3 Occam

Whereas actor languages established the goals for XCPL in terms of expressivity, the programming system Occam in may ways sets the compilation goals for XCPL. Whereas

the XCPL compiler attempts to determine the number of objects and their connectivity for a computation, the Occam compiler expects this information to be supplied by the programmer. Occam works with computational objects called "processes" that are explicitly interconnected by "channels".

Occam recognizes three primitive processes of assignment, input, and output. All other processes are compositions of these three. Processes are combined into larger processes by "constructors". Examples of constructors include SEQ for sequential, PAR for parallel, and ALT for alternate (mutual exclusion) execution. Occam has two conditional constructs called IF and WHILE. A named process in Occam is called a function, but is actually a macro that is expanded at compile time.

Unlike XCPL and actors, Occam includes a notion of channels that are used to connect processes. These channels must be explicitly specified between communicating objects and may not change during the execution of an Occam program. All processes and all channels involved in a computation must be set-up at compile time. To facilitate specification, Occam uses "replicators" to specify multiple instances of a process or channel. The `USING` clause of XCPL corresponds to Occam's replicator, for example, the use of SEQ with a replicator is similar to the `USING` clause with the integer range specified by "..". An important difference is that the arguments supplied to replicators must be expressions comprised *solely* of constants. This restriction is necessary so that the integer values used by the replicators can be computed at compile time.

Occam's lack of a "new" facility to dynamically create processes imposes a severe restriction when writing programs. If a programmer cannot determine the number of processes and channels that a computation will need, then the only recourse is to allocate the maximum number of processes and channels that the system will allow for. The same strategy is used by FORTRAN programmers to support dynamic storage allocation by defining the largest array possible.

The message passing semantics of Occam are substantially different from XCPL's semantics. Occam's send and receive operations are tightly synchronized as opposed to the weak synchronization capabilities of XCPL. Whenever an Occam program encounters a send operation, execution of the process stops until the receiving process has accepted the message. Likewise when a receive is encountered, the process stops until the message arrives. Whenever two processes in Occam jointly perform a send and receive operation, the two processes are synchronized. Occam can select one of many receive operations via the ALT constructor. Except for this one case, every message operation requires the process to stop executing until the message transfer completes.

In comparison to XCPL, Occam resembles an assembly programming language. The specifications for processes that the Occam compiler requires the programmer to supply closely tracks the information that the XCPL compiler attempts to extract by program flow analysis. XCPL does not require that the object structure of a program be fully disclosed at compile time, but rather builds up an initial object graph. If the flow analysis yields a constant graph, then the information available is the same as that for the Occam version. The constructors of PAR and SEQ can be thought of prefix versions of XCPL's comma and period combinators respectively.

4.2.4 Smalltalk

The fundamental difference between XCPL and Smalltalk is that of concurrency. Both languages depend on the metaphor of message passing. In one respect, Smalltalk is simpler

than XCPL because no distinction is made between environment and primitive objects. Thus programmers can write methods for primitive objects. Since for both systems message passing is the only mechanism by which one object can side-effect another, Smalltalk would appear to be a promising starting point for a concurrent language. For example, the period symbol used in Smalltalk block contexts for sequencing could be augmented with XCPL's comma combinator to denote concurrent message passing.

The crucial difference between Smalltalk and XCPL is the accessibility of method names. This difference has a major impact when writing concurrent programs. The name space of Smalltalk is cleanly partitioned between method and variable names. In Smalltalk, names are distinguished by how they are parsed inside a Smalltalk expression. Smalltalk always expects a variable to be immediately followed by a message selector, therefore method names are determined by their *position* within the text string to be parsed. In contrast, XCPL requires that a special character, *i.e.* colon, prefix every method name used as a message selector. For XCPL, potentially any variable name used in an environment object can be a method name.

For Smalltalk, variables can assume any value but methods once assigned a definition can never be altered by a program, only indirectly by the programmer. Thus Smalltalk methods are like XCPL methods that are declared with the equal operator (see Section 2). Smalltalk supports a hierarchical organization scheme for objects called "Classes" and every object belongs to some class. Methods are organized at the class level and every object of a given class shares the same set of methods. For uniprocessor implementations this organization is space-efficient since objects of the same class can share a single "message dictionary". For a concurrent implementation though, multiple copies of a single method dictionary are necessary to prevent a bottleneck from developing when accessing the message dictionary. Once multiple copies of the dictionary are introduced, there is no overriding reason for keeping them consistent, except of course to preserve the semantics of Smalltalk.

For both XCPL and Smalltalk, the arrival of a message at an object causes a method to be executed. For Smalltalk, the method name is looked up in the message dictionary for the class of the object. For XCPL, the variable table local to the object is searched. Given that method activation is intrinsic to message arrival, the question arises as to how to prevent a method from executing under program control. In environments where side-effects occur, such control is needed to ensure that the side-effecting operations are performed in the proper order. A simple example would be a bank account object which holds a balance and reacts to both debit and credit messages. If the balance falls below zero then debit messages should be blocked while still allowing credit messages to get through. In XCPL this action is expressed by setting the debit method name to some value other than a method value. The credit method name is still accessible but the the debit method name has been "withdrawn" from use outside the object.

Since Smalltalk objects cannot alter their method dictionaries, a concurrent Smalltalk system would require either some form of a trigger mechanism [Kotov] to inhibit methods from executing, or require that all queueing be forced upon the objects. The trigger mechanism was used in a concurrent extension to the object language Simula [Lang] by including a new programming construct called "SELECT". Every method in a Simula object could have its own SELECT statement. The SELECT statement accepted a Boolean predicate that was evaluated before the method was invoked. If the predicate evaluated to true then the associated method could be "fired", otherwise no action would be taken until the predicate became true. Another example is Concurrent Smalltalk [Dally] which uses special variables called "locks" that, like the SELECT statement, may be made part

of every method by a "`require..exclude`" statement. Although locks are described for controlling critical sections, the net effect is to order the enabling of methods based on the history of the message flow.

Clearly the issue at hand is how to express the necessary message history for an object. In OCCAM the problem does not exist because of the semantics of the message primitives. The message history for an OCCAM process is simply the trace of statements executed inside the process. XCPL maintains its necessary message history by controlling the binding of names to method definitions. For object languages where the method name space is not accessible, either a trigger mechanism must be included or all queueing be forced upon the objects. For dataflow computations which are by definition history-insensitive, the trigger mechanism is unnecessary; however, for all other cases where the order of message events is important, the message history must be encoded either in the instruction trace, variable bindings, explicit variables, or some other persistent state mechanism.

Smalltalk supports a "becomes" method for class Class that allows the swapping of two object classes. This method, although probably not intended for such an application, could be used to change the available set of methods inside a method dictionary. In practice though, using such a technique effectively would be difficult. In summary, sequential behavior is deeply engrained in the semantics of Smalltalk, although it may be possible to make a concurrent Smalltalk-like system, the transformation is not superficial. Although there are syntactical and metaphorical similarities, XCPL, by using a simpler but less space efficient data structure for representing objects, can obviate many of the problems that a concurrent Smalltalk would have to confront.

4.3 Project Status

To date all the examples have been compiled and tested on the first engine. The plan now is to circulate details of the language to as wide an audience as possible. Work has begun on constructing and testing the later engines proposed in Section 4.1, using the programming examples of Section 3 as test data. More test data for the flow analysis will hopefully be produced by new programs written in XCPL. Any changes to the compilation process will be invisible to the programmer except for differences in runtime performance.

Providing a version of XCPL for Cosmic Cube or iPSC requires both the development of the compiler and a suitable runtime environment. Since the main interest of this research is the compiler, work on the runtime system has been given a lower priority.

Immature poets imitate, mature poets steal.

- T.S. Eliot

5 Acknowledgements

XCPL represents the third installment to the Caltech trilogy on ensemble machines and concurrent programming. The impetus behind all three works, in particular XCPL, has been Chuck Seitz (the "Chuck Yeager" of Computer Science). Thanks are due to Chuck Seitz on many counts, but perhaps most importantly for his inexhaustable supply of thought provoking questions and comments which left no aspect of XCPL untouched.

Thanks are due to Al Davis for many helpful conversations about the challenges of compiling for concurrent machines. Under the auspices of Al Davis, many hours were logged on a Symbolics 3600, which provided the initial testing ground for the programming style that XCPL would be expressly designed for. Thanks are due to Carl Hewitt and Bill Dally for enlightening conversations on what concurrent programs should look like, and to Bob Barton for conversations about the object languages that we really want, but do not know how to formalize yet. Craig Steele performed the impossible task of proofreading and in the process made several insightful comments on the exposition. Finally, thanks are due to A. Cole, Tony Davie, and R. Morrison, for the S-Algol language. Their elegant example of both an Algol derivative and its compiler were instrumental to the implementation of XCPL.

Bibliography

[Agha] G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Artificial Intelligence Laboratory, Technical Report 844, June 1985.

[Baker] H.G. Baker, *Actor Systems for Real-Time Computation*, MIT Artificial Intelligence Laboratory, Technical Report 197, March 1978.

[Baker] H.G. Baker and C. Hewitt, *The Incremental Garbage Collection of Processes*, ACM SIGART-SIGPLAN Symposium, Rochester, N.Y., Aug. 1977.

[Clinger] W.D. Clinger, *Foundations of Actor Semantics*, MIT Artificial Intelligence Laboratory, Technical Report 633, May 1981.

[Dally] W.J. Dally, *Architectures for Concurrent Data Structures*, Dept. of Computer Science, California Institute of Technology, Ph.D. Thesis, Jan. 1986.

[Davie] A.J.T. Davie and R. Morrison, *Recursive Descent Compiling*, Ellis Horwood Limited, Chichester, 1982.

[Guy] R.K. Guy, *How to Factor a Number*, Proceedings of the Fifth Manitoba Conference on Numerical Mathematics, Utilitas Mathematics Publishing Inc., Winnipeg, Oct. 1975.

[Halstead] R.H. Halstead, Jr., *Multilisp: A Language for Concurrent Symbolic Computation*, M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass., July 10, 1985.

[Harrison] M.A. Harrison, *Introduction to Switching and Automata Theory*, McGraw-Hill Book Company, San Francisco, 1965.

[Holstege] E.J. Holstege, *Type Inference in a Declarationless, Object-Oriented Language*, Dept. of Computer Science, California Institute of Technology, Technical Report 5035, June 1982.

[Ingalls] D.H. Ingalls, *The Smalltalk 76 Programming System: Design and Implementation*, Proceedings of the Fifth ACM Conference on Principles of Programming Systems, pp. 9-16, January 1978.

[Inmos] Inmos Limited, *IMS T424 Reference Manual*, Order No. 72 TRN 006 00, Bristol, United Kingdom, Nov. 1984.

[Intel] Intel Scientific Computers, *iPSC User's Guide*, Order No. 175455-001, 15201 N.W. Greenbrier Parkway, Beaverton, Oregon, Aug. 1985.

[Kotov] J. Miklosko and V.E. Kotov, *Algorithms, Software and Hardware of Parallel Computers*, VEDA, Publishing House of the Slovak Academy of Sciences, Bratislava, 1984.

[Kennedy] K. Kennedy, "A Survey of Data Flow Analysis Techniques", *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1981.

[Lang] C.R. Lang, Jr., *The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture*, Dept. of Computer Science, California Institute of Technology, Technical Report 5014, May 1982.

[McEliece] R.J. McEliece, *The Theory of Information and Coding*, Addison-Wesley Publishing Co., Reading, Mass., 1977.

[Muchnick] S. Muchnick and N. Jones, "Flow Analysis and Optimization of LISP-Like Structures", *Program Flow Analysis: Theory and Applications*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[Padua] D.A. Padua, D.J. Kuck and D.H. Laurie, *High-Speed Multiprocessors and Compilation Techniques*, IEEE Transactions on Computers, C-29, pp. 763-776, 1980.

[CPGCC] W-K. Su, R. Faucette and C.L. Seitz, *C Programmer's Guide to the Cosmic Cube*, Dept. of Computer Science, California Institute of Technology, Technical Report 5203, Sept. 1985.

[Seitz83] Charles Lewis Seitz, *Experiments with VLSI Ensemble Machines*, Dept. of Computer Science, California Institute of Technology, Technical Report 5102, Oct. 1983.

[Seitz85] Charles Lewis Seitz, *The Cosmic Cube*, Communications of the ACM, Vol. 28, No. 1, pp. 22-33, Jan. 1985.

[Steele] C.S. Steele, *Placement of Communicating Processes on Multiprocessor Networks*, Dept. of Computer Science, California Institute of Technology, Technical Report 5184, Apr. 1985.

[Theriault] D.G. Theriault, *Issues in the Design and Implementation of Act2*, MIT Artificial Intelligence Laboratory, Technical Report 728, June 1983.

[Wirth] N. Wirth and H. Weber, *EULER: A Generalization of ALGOL, and its Formal Definition: Part II*, Communications of the ACM, Vol. 9, No. 2, pp. 89-99, Feb. 1966.